Slice Sampling for Probabilistic Programming

Razvan Ranca
University of Cambridge

Zoubin Ghahramani
University of Cambridge

Abstract

We introduce the first, general purpose, slice sampling inference engine for probabilistic programs. This engine is released as part of StocPy, a new Turing-Complete probabilistic programming language, available as a Python library. We present a transdimensional generalisation of slice sampling which is necessary for the inference engine to work on traces with different numbers of random variables. We show that StocPy compares favourably to other PPLs in terms of flexibility and usability, and that slice sampling can outperform previously introduced inference methods. Our experiments include a logistic regression, HMM, and Bayesian Neural Net.

1 Introduction

There has been a recent surge of interest in probabilistic programming, as demonstrated by the continued development of new languages (eg: Wood et al. [2014], Goodman et al. [2008], Lunn et al. [2009], Milch et al. [2007]) and by the recent DARPA[1] program encouraging further research in this direction. The increase in activity is justified by the promise of probabilistic programming, namely that of disentangling model specification from inference. This abstraction would open up probabilistic modelling to a much larger audience, including domain experts, while also making model design cheaper, faster and clearer.

Two of the major challenges lying ahead for Probabilistic Programming Languages (PPLs), before their full promise can be achieved, are those of usability and inference performance (Gordon [2013]). In this paper we address usability by presenting a new PPL, StocPy, available online as a Python library (Ranca [2014]). We also take a step towards better inference performance by implementing a novel, slice sampling, inference engine in StocPy.

StocPy is based on the PPL design presented by Wingate et al. [2011], but is written purely in Python, and works on model definitions written in the same language. This enables us to take full advantage of Python's excellent prototyping abilities and fast development cycles, and thus allows us to specify models in very flexible ways. For instance models containing complex control flow and elements such as stochastic (mutual) recursion are easily representable. Additionally, the pure Python implementation means StocPy itself provides a good base for further experiments into PPLs, such as defining novel naming techniques for stochastic variables, looking at program recompilation to improve inference performance, or testing out new inference engines. We illustrate the benefits StocPy offers by discussing some of the language's features and contrasting the definitions of several models in StocPy against those in Anglican (Wood et al. [2014]).

We believe that ease of prototyping and implementing novel inference engines is crucial for the future success of PPLs. As decades of compiler design have shown , a "magic bullet" inference technique is unlikely to be found (eg: [Appel, 1998, p. 401]). This is re-inforced by the fact that PPLs can allow for a great deal of flexibility regarding the types of models and queries posed (eg: propositional logic can be represented by exposing delta functions, probabilistic programs can be conditioned on arbitrary boolean expressions). Such flexibility means a general PPL inference engine has a daunting task ahead, which includes subtasks such as handling boolean satisfiability problems. Therefore, it seems vital to develop a toolbox of inference techniques which work well on different types of modelling tasks. This high-level reasoning supports not only the development of StocPy itself, but also of the slice sampling inference engine, which outperforms previous inference techniques on certain classes of models.

[1] Probabilistic Programming for Advancing Machine Learning: http://ppaml.galois.com/wiki/

Preliminary work. Under review by AISTATS 2014. Do not distribute.

```
1 import stocPy
2
3 def guessMean():
4   mean = stocPy.normal(0, 1, obs=True)
5   stocPy.normal(mean, 1, cond=5)
6
7 samples = stocPy.getSamples(guessMean, 10000, alg="slice")
8 stocPy.plotSamples(samples, xlabel="Mean")
```

Figure 1: Complete StocPy program that infers a gaussian's mean.

2 StocPy

We implement a novel PPL, StocPy, which is made available online (Ranca [2014]). StocPy has both a Metropolis-Hastings inference engine (implemented in the style presented by Wingate et al. [2011]) and a Slice sampling inference engine which we further discuss in Section 3. Weighted combinations of the two inference strategies can also be used.

2.1 Why make a Python PPL?

One of the main promises of PPLs is to allow quick and cheap development of probabilistic models, including by domain experts who may not be very comfortable with machine learning, or even with programming. The popularity of PPLs such as BUGS is partly due to their simplicity. On the other hand, lisp-based PPLs such as Goodman et al. [2008] offer greater flexibility but at the price of usability. StocPy is able to offer the same power as the lisp-based PPLs while also allowing the user to work in Python, a language both very friendly to beginners and extremely popular for prototypying.

Additionally, using Python means we can make StocPy easily available as a library, and that we can make use of Python's flexible programming style when defining models (eg: stochastic mutually recursive functions, use of globals). A user need only provide a function entry point to their model, which can then call any other functions, make use of any globals defined in the file, and use any common[2] Python constructs as needed.

Finally, using Python offers both the user and the language designer a rich library support. For instance, as language designers, we are able to provide support for all 96 stochastic primitives defined in the "scipy.stats" library[3] by defining a single wrapper function. Without this library, all 96 primitives would have had to be specified, in a computationally efficient way, in the language itself, which would have required a significant amount of effort. The library support also reduces the barrier of entry for all manner of PPL research. For instance, Python's excellent network and graph libraries (eg: Hagberg et al. [2008]) could be used to define a novel naming convention for stochastic primitives, which takes the program's control flow into account. Such a naming convention is required by the framework of Wingate et al. [2011] and could be an improvement over the simpler version presented in that paper. In general, StocPy tries to accomodate for such avenues of research (eg: StocPy's automatic variable naming can be turned off by setting a single flag).

2.2 StocPy Programming Style

As mentioned above, a user defined model can make use of most common Python features, as long as the model is confined to one file. The interaction with StocPy is meant to be lightweight. Essentially, the user should perform all stochastic operations via StocPy, rather than another library of random numbers. That is to say, rather than calling `scipy.stats.norm(0,1)`, the user could call either the StocPy stochastic primitive directly: `StocPy.normal(0,1)`[4], or use the StocPy wrapper over scipy.stats: `StocPy.stocPrim("normal", (0,1))`. By defining stochastic primitives through StocPy, the user will define a valid generative model. Conditioning is done within the same function call that defines a variable, by specifying the "cond" parameter (eg: `StocPy.normal(0,1,cond=True)`). Finally, the variables or expressions we'd like to observe can be specified in two ways, either directly, in the variable definitions, via the "obs" parameter (eg: `stocPy.normal(0,1,obs=True)`), or via a bespoke observe statement (eg: `StocPy.observe(expression, "expName")`). The later is more flexible, allowing us to observe the result of arbitrary expressions aggregated, via their name, in arbitrary ways. Once the model is defined, inference can be done by calling one of several inference functions, and passing the entry point to the model as a parameter. A minimal (but complete) model definition, which tries to infer the mean of a gaussian given a single observation, is shown in Figure 1. This model is revisited in Figures 10 and 9 where we see its true posterior and abstract model specification respectively.

In Figure 1, line 1 imports our library. Line 3 de-

[2] More exotic constructs, such as the use of list comprehensions in place of for loops, are not currently supported by the automatic variable naming. However they can be used, if the stochastic primitives involved are manually named by the user.

[3] scipy.stats primitives: http://docs.scipy.org/doc/scipy/reference/stats.html

[4] these are only availalble for a few distributions. See the StocPy library for details

fines the function that is the entry point to our model. Line 4 specifies the prior on our mean (also a gaussian) and tells StocPy that we want to observe the values this variable takes. Line 5 conditions a normally distributed random variable with our sampled mean and variance 1, on our only observation (5). Line 7 performs inference on our model, extracting 10,000 samples via slice sampling. Finaly, line 8 uses a StocPy utility function to plot the resulting distribution, which is shown in Figure 4.

In Figures 2, 3 we present more examples of models expressed in StocPy and in Anglican. The models are 2 of those presented by Wood et al. [2014] (more models are included in the supplementary material). The Anglican specifications are taken either directly from the paper or from the Anglican website[5]. We can see that in the case of the more complex models, we are able to provide a more succint representation in StocPy than in Anglican.

3 Slice Sampling Inference Engine

3.1 Advantages of Slice Sampling

Slice sampling (Neal [2003]) is a Markov Chain Monte Carlo algorithm that can extract samples from a distribution $P(x)$ given that we have a function $P^*(x)$, which we can evaluate, and which respects $P^*(x) = ZP(x)$, for some constant $Z > 0$. The idea behind Slice sampling is that, by using an auxiliary height variable u, we can sample uniformly from the region under the graph of the density function of $P(x)$.

In order to get an intuition of the algorithm we can consider the one dimensional case. Here we can view the algorithm as a method of transitioning from a point (x, u) which lies under the curve $P^*(x)$ to another point (x', u') lying under the same curve. The addition of the auxiliary variable u means that we need only sample uniformly from the area under the curve $P^*(x)$.

A basic Slice sampling algorithm can be described as:

Algorithm 1 Slice Sampling

1: Pick initial x_1 such that $P^*(x_1) > 0$
2: Pick number of samples to extract N
3: **for** $t = 1 \rightarrow N$ **do**
4: Sample a height u uniformly from $[0, P^*(x_t)]$
5: Define a segment $[x_l, x_r]$ at height u
6: Pick $x_{t+1} \in [x_l, x_r]$ such that $P^*(x_{t+1}) > u$

The key to Slice sampling's efficiency is the fact that the operations in lines 5 and 6 can be performed with exponential stepping out and shrinking methods. One possible implementation of these operations is shown in the supplementary material. In this way, slice sampling corrects one of the main disadvantages of MH, namely that it has a fixed step size. In MH, picking a wrong step size can significantly hamper progress either by unnecesarily slowing down the random walk's progress or by resulting in a large proportion of rejected sampled. Slice sampling, however, adjusts an inadequate step size of size S with a cost that is only logarithmic in the size of S (MacKay [2003]).

While some methods to mitigate MH's fixed step size exist, such as performing pre-run tests to determine a good step-size, these adjustment are not possible in the variant of MH commonly used in PPLs. We refer to the MH proposal described in Wingate et al. [2011], which is the same as the "RDB" benchmark used in Wood et al. [2014]. In this MH implementation, whenever we wish to resample a variable, we run the model until we encounter the variable and then sample it from its prior conditioned on the variables we have seen so far in the current run. Therefore no fine-tuning of the proposal distribution is possible.

To get an idea of what can be gained with slice sampling over simple, single-site, MH we look at two examples, one favourable to MH and one unfavourable. In this way we can get an idea of the expected "best" and "worst" case performances of the two. We choose to look at a simple gaussian mean inference problem, and place different priors over the mean. The best case for MH is a prior that is identical to the posterior, which means MH as described above (i.e. RDB) is already sampling from the correct distribution. Conversely, the worst case for MH is an extremely uninformative prior, such that most samples will end up being rejected (in our example, the prior is uniform between 0 and 10,000 while the posterior is very close to a normal with mean 2 and variance 0.032, more details in the supplementary material).

The results are presented in Figures 5, 6. Here we report the Kolmogorov Smirnov (KS) distance between the analytically derived posterior and a running average of the inferred posterior. The x axis shows the number of times the model has been interpreted (and had it's trace likelihood computed). We run the tests multiple times starting from different random seeds and report both the median (solid line) and the 25% and 75% percentiles (dashed lines) over runs. In this experiment we can see that the potential upside of slice sampling overshadows the downside. In the worst case scenario for slice sampling, we have an inference problem where our prior is equal to the posterior. That is we already know the correct answer before seeing any

[5] Anglican model repository:
http://www.robots.ox.ac.uk/∼fwood/anglican/examples/

```
def branching():
  r = stocPy.poisson(4, obs=True)
  l = 6 if r > 4 else fib(3*r) + stocPy.poisson(4)
  stocPy.poisson(l, cond=6)
```

```
[assume r (poisson 4)]
[assume l (if (< 4 r) 6 (+ (fib(* 3 r)) (poisson 4)))]
[observe (poisson l) 6]
[predict r]
```

Figure 2: Branching model expressed in StocPy (left) and Anglican (right)

```
sProbs = (1.0/3, 1.0/3, 1.0/3)
tProbs = {0 : (0.1, 0.5, 0.4), 1 : (0.2, 0.2, 0.6), 2 :
     (0.15, 0.15, 0.7)}
eMeans = (-1,1,0)

def hmm():
  states = []
  states.append(stocPy.categorical(sProbs,obs=True))
  for ind, ob in stocPy.readCSV("hmm-data.csv"):
    states.append(stocPy.categorical(tProbs[states[ind]], obs
         =True))
    stocPy.normal(eMeans[states[ind]], 1, cond=ob)
```

```
[assume initial-state-dist (list (/ 1 3) (/ 1 3) (/ 1 3))]
[assume get-state-transition-dist (lambda (s) (cond ((= s 0)
     (list 0.1 0.5 0.4)) ((= s 1) (list 0.2 0.2 0.6)) ((= s
     2) (list 0.15 0.15 0.7))))]
[assume transition (lambda (prev-state) (discrete (
     get-state-transition-dist prev-state)))]
[assume get-state (mem (lambda (index) (if (<= index 0) (
     discrete initial-state-dist) (transition (get-state (-
     index 1))))))]
[assume get-state-obs-mean (lambda (s) (cond ((= s 0) -1) ((=
      s 1) 1) ((= s 2) 0)))]
[observe-csv "hmm-data.csv" (normal (get-state-obs-mean (
     get-state $1)) 1 ) $2]
[predict (get-state 0)]
[predict (get-state 1)]
...
[predict (get-state 16)]
```

Figure 3: HMM model expressed in StocPy (left) and Anglican (right)

data. In this case the extra overhead incurred by Slice sampling slows it down roughly by a factor of 2 when compared to MH (Figure 5). In the second case, however, we have a very uninformative prior and here slice sampling significantly outperforms MH (Figure 6). In fact, the difference between the 2 algorithms will only get more pronounced as the prior gets more uninformative. That is, examples can be created, where Slice sampling is arbitrarily faster than MH. However, we must nore that these examples are of very simple, one dimensional, unimodal models. The generalization of the insights gained from these examples to more complex models is non-trivial. Even so, we have shown a category of models where slice sampling performs substantially better than MH. Comparisons on more complex models are carried out in Section 4.

3.2 Inference engine construction

Our engine follows the basic slice sampling algorithm presented in Algorithm 1. As presented in Wingate et al. [2011], we consider a sample x to be a program execution trace and $P^*(x)$ to be the likelihood of all stochastic variables sampled in trace x.

The bottleneck in the inference engines is the trace likelihood calculation. Metropolis calculates this value exactly once per sample whereas Slice sampling needs to calculate it at least 3 times (one each for x_l, x_r and the next x), and potentially many more times. For this reason, StocPy provides the possibility to do inference until a certain number of trace log likelihood calculations have been performed, which allows us to compare Metropolis and slice sampling directly and fairly. Further, all experiments comparing slice with Metropolis ran the algorithms for the same length of time. We however chose to use trace likelihoods rather than seconds on the x-axis, which implicitely shows that the two engines average the same number of trace-likelihood calculation per unit of time.

The main non-trivial aspect, and novel contribution, of the inference engine construction is handling trans-dimensional models. We discuss this below.

3.2.1 Trans-dimensional models

In order to understand the additional complications that trans-dimensional models present for inference engines we look at a simple example taken from Wood et al. [2014], the Branching model. This model has 2 variables whose values determine the distribution of a 3rd variable which we condition on. This model is trans-dimensional since on different traces either 1 or both of the 2 variables will be sampled.

Re-writing the model so that both variables are always sampled, even if one of them is unused, leaves the posterior invariant. Therefore one method to correctly perform inference in a trans-dimensional model is to always sample all variables that might be used in a trace. This approach will however be extremely inefficient in large models and is not a viable general solution. In Figure 7 we use this trick to see what the space of possible trace likelihoods looks like, and what the true posterior is. Here pois1 and pois2 refer to the Poisson variables sampled in lines 2 and 3, respectively, of the StocPy model shown in Figure 2. Integrating out the pois2 variable from the above trace likelihood space results in the correct posterior distribution.

The issue with trans-dimensional jumps comes from the fact that a naive inference algorithm will not sam-

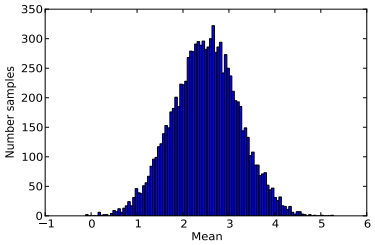

Figure 4: Inferred posterior of the model shown in Figure 1

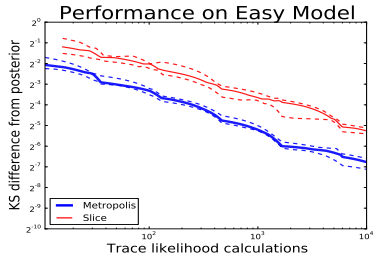

Figure 5: Case most favourable to MH, where the prior and the posterior are both equal to Normal(0, 1)

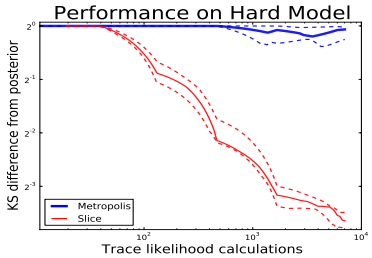

Figure 6: Case which is unfavourable to MH, where the prior is Uniform(0, 10000) and the posterior is roughly Normal(2, 0.032)

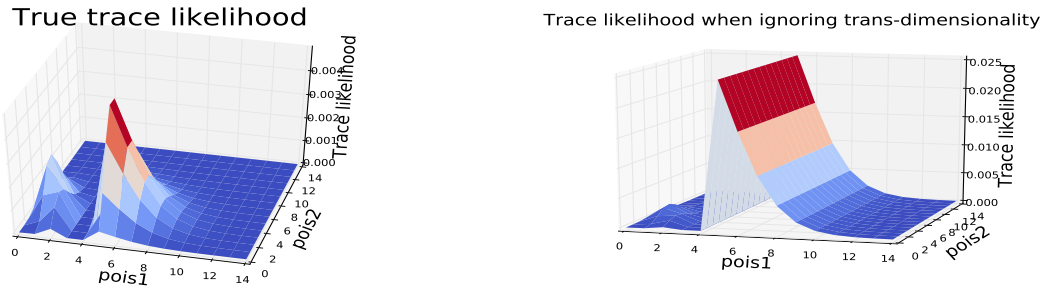

Figure 7: Likelihood space of the execution traces

ple the second poisson when it is not necessary, but will still think that the trace likelihoods of runs with different numbers of sampled variables are comparable. In doing so, the inference engine will be pretending to be sampling from a 2D trace likelihood even when it really is 1D. The space of likelihoods implied by the naive inference engine, and the posterior it would obtain by integrating out pois2, is shown in Figure 8. We now describe how to correct the slice sampling inference engine to handle trans-dimensional models.

3.3 Transdimensional Slice Sampling

To understand the trans-dimensional corrections we can place them in the framework of Reversible Jump Markov Chain Monte Carlo (RJMCMC, Green and Hastie [2009]). Informally, we can think of slice sampling as a form of RJMCMC in which we carefully choose the next sample so that the acceptance probability will always be 1. We include a short explanation of the RJMCMC notation in the Appendix.

In order to place slice sampling in this framework, we can think of a program execution trace as a state x. The choice of move we make is equivalent to choosing which random variable we wish to resample. Therefore, if there are $|D|$ random variables present in the current trace, and we pick one to sample uniformly, then $j_m(x) = 1/|D|$. Once the variable is chosen, we can define a deterministic function h_m which produces a new value for variable m by following the slice sampling specification presented in lines 4-6 of Algorithm 1. The randomness that h_m must take as input is $u \sim g_m$, where u is composed of r random numbers and g_m is the joint distribution of these r numbers. The probability of moving from state x to state x' is then $\pi(x)g_m(u)/|D|$, and the probability of going back from x' to x is $\pi(x')g'_m(u')/|D'|$. Intuitively $\pi(x) = \pi(x')$ since slice sampling samples uniformly under its target distribution. Transdimensionality means that the number of variables sampled in traces will be different and therefore that D' will be different from D and that the dimensionality of u and u' (r and r' respectively) will also be different.

The correction we aplly to account for this is similar to the one applied by Wingate et al. [2011]. Specifically we update $P^*(x) = P^*(x) + log\left(\frac{|D|*p_{stale}}{|D'|*p_{fresh}}\right)$, where p_{stale} is the probability of all variables that are in the current trace but not the new and p_{fresh} is the probability of variables sampled in the new trace that are not in the current.

4 Empirical Evaluation

4.1 Inferring the mean of a Gaussian

We begin with the inference of the mean of a gaussian. This time the prior and posterior are of similar

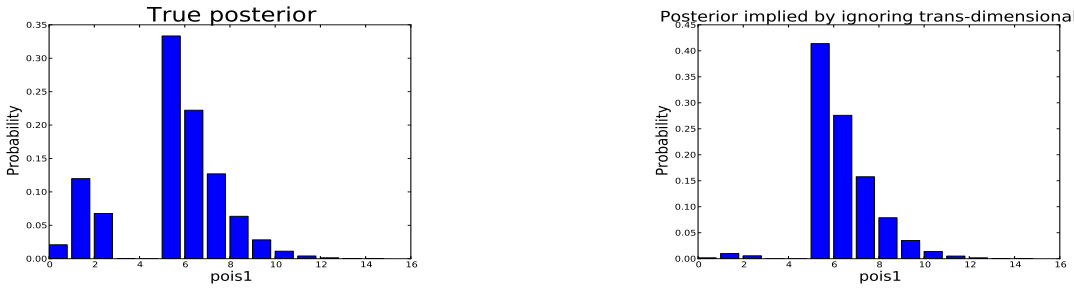

Figure 8: Posteriors inferred by integrating out pois2

sizes, but the posterior is shifted by an unusual observation. In this setting we look at 3 models, namely 1-dimensional, 2-dimensional and trans-dimensional. The model specifications are given in Figure 9 and the models' posteriors are shown in Figure 10.

We now look at the performance of Metropolis, slice sampling and some different mixtures of the two over the 3 models. The mixture methods work by flipping a biased coin before extracting each sample in order to decide which inference method to use.

To compare the inference engines, we extract samples until a certain number of trace likelihood calculations are performed and then repeat this process 100 times, starting from different random seeds. We then plot the median and the quartiles of the KS differences from the true posterior, over all generated runs. Figure 11 shows the quartiles of the runs on the three models.

On the simple, 1d, model all variants of Slice sampling clearly outperform Metropolis. In the quartile graph we consider mixtures of Metropolis and Slice both with 10% Metropolis and with 50% Metropolis and find that the change doesn't have a significant impact on performance. This is likely because, if slice picks a good sample, Metropolis is likely to simply keep it unchanged (since it will tend to reject the proposal from the prior if it is worse).

On the 2d model, slice still clearly outperforms Metropolis, though the gap is not as pronounced as for the 1d model. Further, as in the 1d model, the 3 different slice variants all get quite similar performance. Additionally, on this model, the fact that the slice mixtures get more samples per trace calculation translates into a slightly better performance for them than for the pure slice sampling method.

The third, trans-dimensional, model reveals a more pronounced performance difference between slice and Metropolis than in the 2 dimensional case. Further, on this model, we see a significant gap between the 1:9 Metropolis:Slice method and the other slice sampling approaches. It seems that, in this case, the ratio of 1:9 strikes a good balance between slice sampling's automatic adjustment of the kernel's width and Metropolis' efficiency in likelihood calculations per sample.

4.2 Anglican models

In order to further test the Slice sampling inference engine we look at 3 of the models defined in Wood et al. [2014]. The model specifications for these are provided in Section 2.2 and in the supplementary material.

To evaluate the engines, we use each to generate 100 independent sample runs. In Figure 12 we plot the convergence of the empirical posterior to the true posterior as the number of trace likelihood calculations increase. For continuous distributions we use the Kolmogorov-Smirnov statistic (as before), whereas for discrete ones we employ the Kullback-Leibler divergence to measure the similarity of two distributions.

On the Branching and HMM models (12) naive Metropolis-Hastings outperforms all Slice combinations. These models are quite small and are conditioned on few datapoints, which means they have relatively similar priors and posteriors. On such models, the overhead of slice sampling cannot be recovered, since the naive Metropolis actually does quite well by simply sampling the prior. Additionally, in the HMM model, we are only inferring in which of 3 states the model is in. With such a small state space the value that an adjusting proposal kernel can add is limited.

On the Marsaglia model however (12), slice sampling does obtain a boost in performance. This is due both to the continuous nature of the model (i.e. larger state space) and to the fact that the prior and posterior are significantly different (figure in Appendix). It's worth noting that the Marsaglia model is the only one in which Metropolis from the prior outperformed Anglican's Particle MCMC. Slice sampling is therefore better than both PMCMC and Metropolis on this model.

It is interesting to note that, on all models tried, slice outperforms Metropolis on a per-sample basis. This is an unfair comparison since slice does more "work" to generate a sample than Metropolis, but it illustrates the point that the samples chosen by slice are in gen-

$NormalMean1$:
$m \sim N(0, 1)$
observe $N(m, 1) = 5$
predict m

$NormalMean2$:
$m \sim N(0, 1)$
$v \sim invGamma(3, 1)$
observe $N(m, v) = 5$
predict m

$NormalMean3$:
$m \sim N(0, 1)$
if $m < 0$: $v \sim invGamma(3, 1)$
else: $v = 1/3$
observe $N(m, v) = 5$
predict m

Figure 9: Specifications of the 3 gaussian mean inference models

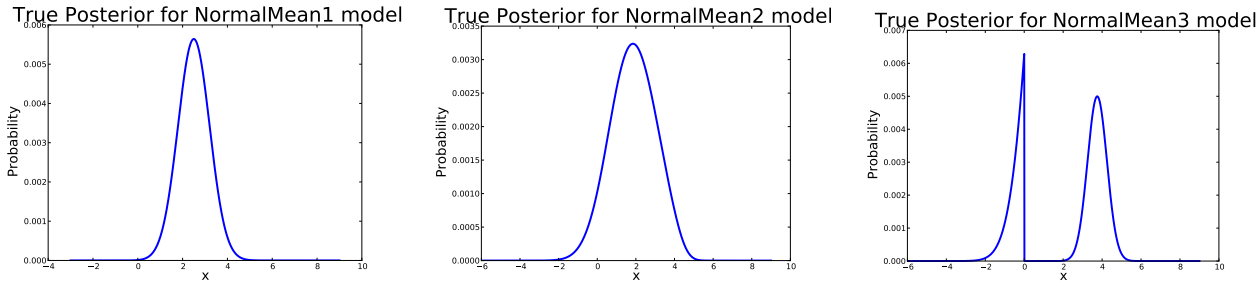

Figure 10: Analytically derived posteriors of the 3 gaussian mean inference models

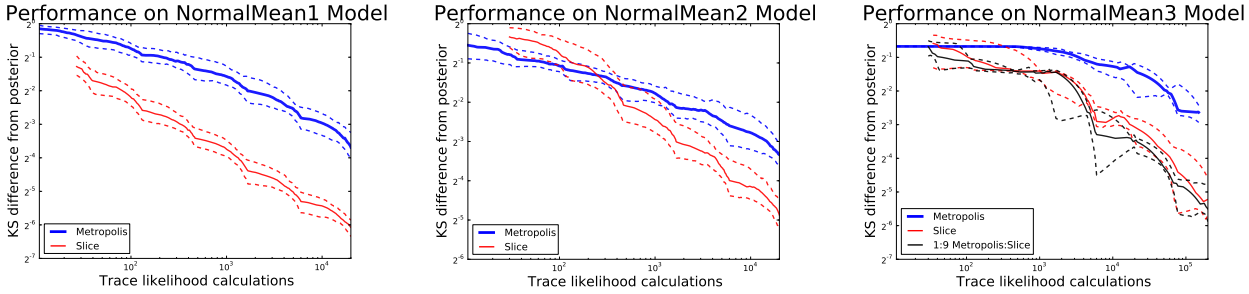

Figure 11: Median (solid), 25% and 75% quartiles (dashed) convergence rates on the 3 Gaussian models

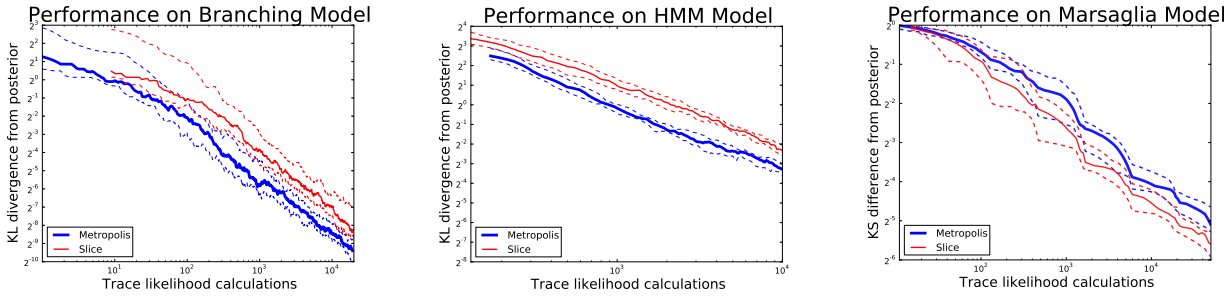

Figure 12: Median (solid), 25% and 75% quartiles (dashed) convergence rates on the 3 Anglican models

eral better, the only question is if the difference is sufficient as to justify the increased overhead.

4.3 Models for classification

Lastly we look at some new models, namely logistic regression and a small neural network. Since we are doing classification, here we can actually plot the mean squared error that the model achieves after a certain number of trace likelihood computations. As before, we perform separate runs starting with different random seeds and plot both the median and the 25% and 75% quartiles.

First we perform logistic regression on the well known Iris dataset, obtained from Bache and Lichman [2013]. The result is shown in Figure 13 and shows that Slice sampling converges significantly faster than MH. Secondly we train a small Bayesian neural network on the same dataset. Our neural network is composed of an input layer of the same dimensionality as the data (4), two hidden layers of 4 and 2 neurons respectively and an output layer of a single neuron (since we are treating the task as a binary classification problem). In Figure 14 we see that slice sampling does significantly better than the MH baseline on this task. We also notice that the harder the inference problem is, the more the margin by which slice sampling outperforms grows. Iris-setosa, on which the performance gap is smallest,

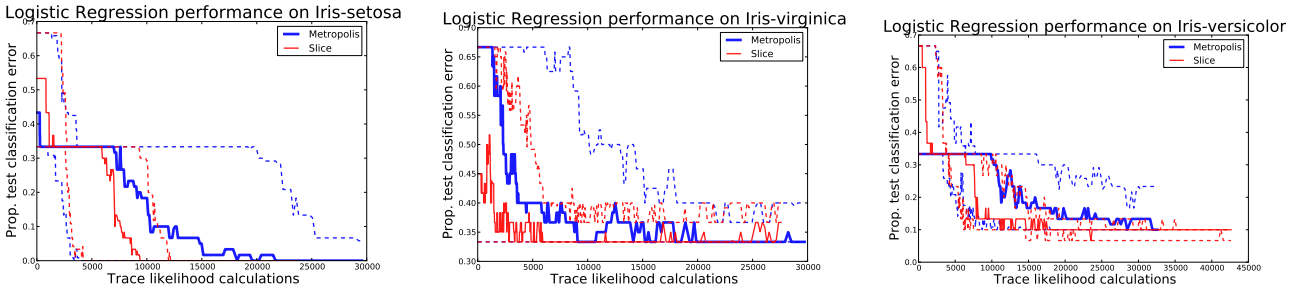

Figure 13: Median (solid), 25% and 75% quartiles (dashed) convergence rates of logistic regression on Iris dataset

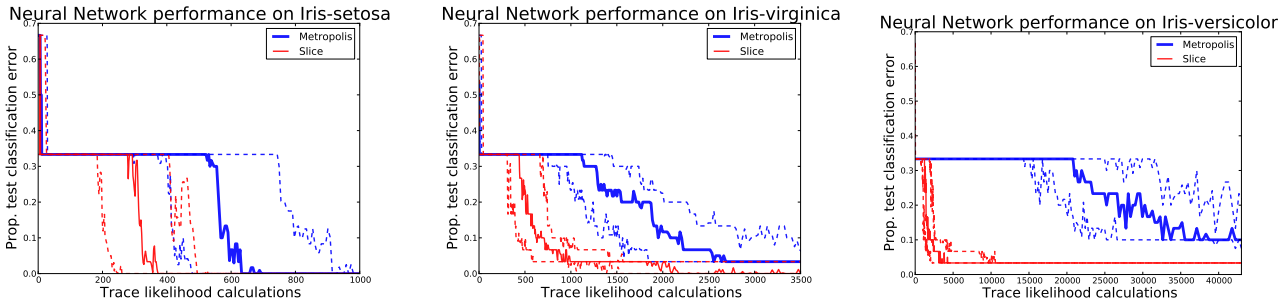

Figure 14: Median (solid), 25% and 75% quartiles(dashed) convergence rate of a Bayesian NN on the Iris dataset

is linearly separable from the other 2 classes, which are not linearly separable from one another. In the case of Iris-versicolor, we were not able to run the engine long enough for Metropolis to match slice's performance, as Metropolis is still lagging behind even after running 10 times longer than slice.

5 Related Work

Our slice sampling inference engine is based on the slice sampling method presented by Neal [2003], and influenced by the computer friendly slice sampler shown in MacKay [2003].

Slice sampling techniques have been applied to a wide range of inference problems (Neal [2003]) and are used in some of the most popular PPLs, such as BUGS (Lunn et al. [2009]) and STAN (Stan Development Team [2014]). However, the slice samplers these languages employ are not exposed directly to the user, but instead only used internally by the language. Therefore, the slice samplers present in these languages are not intended to generalise to all models and, specifically, make no mention of trans-dimensionality corrections. Poon and Domingos [2006], also proposes a slice sampling based solution, this time to Markov Logic problems. However this algorithm is focused on deterministic or near-deterministic inference tasks and so bears little resemblance to our approach.

The most similar use-case to ours would be in Venture (Mansinghka et al. [2014]). Here however slice sampling is only mentioned in passing, amongst other inference techniques the system could support. No details, nor discussions of trans-dimensionality, are present.

6 Conclusion

We have introduced and made available StocPy, the first general purpose Python Probabilistic Programming Language. We have shown the benefits StocPy offers both to users and designers of PPLs, namely flexibility, clarity, succintness and ease of prototyping.

We have also implemented a novel, slice sampling, inference engine in StocPy. To our knowledge this is the first general purpose slice sampling inference engine and the first slice sampling procedure that solves the problem of trans-dimensionality.

We have empirically evaluated this slice sampling engine and shown that the potential benefits far outweigh the potential costs, when compared to single-site Metropolis. While Metropolis works well on very small models where the prior and the posterior are similar, slice provides substantial benefits as the distributions diverge. Additionally, on the models where Metropolis performs best slice only experiences a constant slowdown due to its overhead, whereas when Metropolis performs poorly the performance difference can be arbitrarily large.

Finally, we have provided comparisons with Anglican which show promising results despite slice sampling not beeing optimised for runtime speed. A full benchmarking of the systems remains to be performed.

References

A. W. Appel. *Modern compiler implementation in ML*. Cambridge university press, 1998.

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008*, pages 220–229, 2008.

A. D. Gordon. An agenda for probabilistic programming: Usable, portable, and ubiquitous. 2013.

P. J. Green and D. I. Hastie. Reversible jump MCMC. *Genetics*, 155(3):1391–1403, 2009.

A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, Aug. 2008.

D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistics in medicine*, 28(25):3049–3067, 2009.

D. J. MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. *Statistical relational learning*, page 373, 2007.

R. M. Neal. Slice sampling. *Annals of statistics*, pages 705–741, 2003.

H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*, volume 6, pages 458–463, 2006.

R. Ranca. StocPy: An expressive probabilistic programming language, in python. `https://github.com/RazvanRanca/StocPy`, 2014.

Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.4*, 2014. URL `http://mc-stan.org/`.

D. Wingate, A. Stuhlmueller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.

Supplementary Material for "Slice Sampling for Probabilistic Programming"

Razvan Ranca
University of Cambridge

Zoubin Ghahramani
University of Cambridge

1 Detailed slice-sampling pseudocode

Here we present the detailed pseudocode for one possible slice sampling implementation. Algorithm 1 is the high-level logic of slice sampling which was also presented in the paper.

The other algorithms expand upon this, showing the lower level operations. Specifically, Algorithm 2 shows how we could perform the operation described in line 5 of Algorithm 1, by exponentially increasing the $[x_l, x_r]$ interval. Similarly, Algorithm 3 describes how the operation from line 6 of Algorithm 1 is performed, namely by sampling a next x placed on the already defined segment $[x_l, x_r]$ such that x is choosen "uniformly under the curve".

Algorithm 1 Slice Sampling

1: Pick initial x_1 such that $P^*(x_1) > 0$
2: Pick number of samples to extract N
3: **for** $t = 1 \rightarrow N$ **do**
4: Sample a height u uniformly from $[0, P^*(x_t)]$
5: Define a segment $[x_l, x_r]$ at height u
6: Pick $x_{t+1} \in [x_l, x_r]$ such that $P^*(x_{t+1}) > u$

Algorithm 2 Return appropriate $[x_l, x_r]$ interval, given current sample x and its height u

1: **function** StepOut(x, u)
2: Pick small initial step width w_i
3: **while** $P^*(x - w) < u$ or $P^*(x + w) < u$ **do**
4: $w_i = w_i/2$
5: $w = w_i$
6: **while** $P^*(x - w) > u$ **do**
7: $w = w * 2$
8: $x_l = x - w$
9: $w = w_i$
10: **while** $P^*(x + w) > u$ **do**
11: $w = w * 2$
12: $x_r = x + w$
13: **return** x_l, x_r

Algorithm 3 Given current sample x and its height u, get next sample x_n from the interval $[x_l, x_r]$ such that $P^*(x_n) > u$

1: **function** SampNext(x, u, x_l, x_r)
2: Sample x_n uniformly from $[x_l, x_r]$
3: **while** $P^*(x_n) \leq u$ **do**
4: **if** $x_n > x$ **then**
5: $x_r = x_n$
6: **else**
7: $x_l = x_n$
8: Sample x_n uniformly from $[x_l, x_r]$
9: **return** x_n

2 True Posteriors

Here we present 2 true posteriors that support some claims made in the paper. In Figure 1, we see that the "Hard" Gaussian mean inference problem results in an analytical posterior that is very close to a Gaussian with mean 2 and standard deviation 0.032. The second posterior, shown in Figure 2 presents the difference between the prior and posterior of the Marsaglia model. This figure gives an intuitive understanding of why Metropolis is outperformed by slice sampling on the Marsaglia model. It is easy to imagine how bad Metropolis will do in this model if it only samples from the prior.

3 Additional Sample Programs in StocPy and Anglican

For completeness and as to get a better idea of the "feel" of StocPy we show the remaining 2 Anglican models that were not presented in the paper. These models are a Dirichlet Process Mixture (Figure 3) and a Marsaglia (Figure 4). We also show these models' implementation in Anglican, for comparison's sake.

As in the paper, we can see that on the more complex model StocPy manages to be more succint, largely thanks to the Python in-built operators. Additionally,

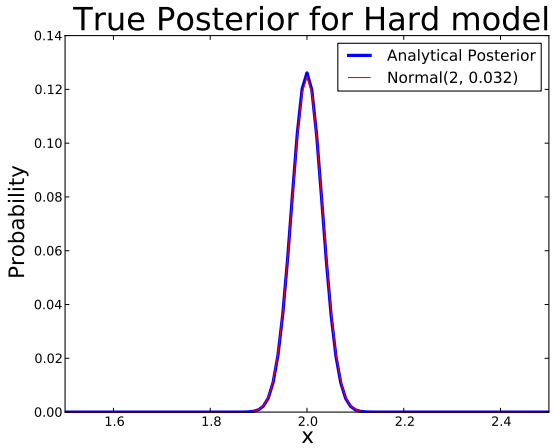

Figure 1: Analytically derived posterior for the Hard normal model, superimposed over a Gaussian

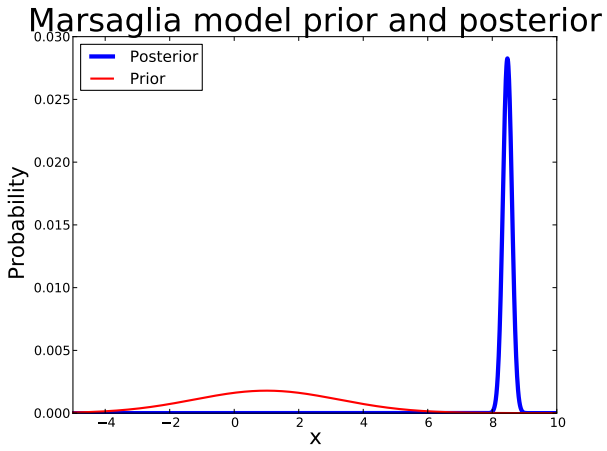

Figure 2: The posterior on the Marsaglia model is sufficiently different from the prior for slice sampling to outperform Metropolis

```
def dpMix():
  crp = stocPy.CRP(1.72)
  sds, ms, cs = {}, {}, {}
  for ind, ob in stocPy.readCSV("dpm-data.csv"):
    c = crp(ind)
    cs[ind] = c
    if c not in ms:
      sds[c] = math.sqrt(10 * stocPy.stocPrim("
          invgamma", (1,0,10)))
      ms[c] = stocPy.stocPrim("normal", (0, sds[c]))
    stocPy.normal(ms[c], sds[c], cond=ob)
  stocPy.observe(cs, name="u")
  stocPy.observe(len(ms), name="K")
```

```
[assume class-generator (crp 1.72)]
[assume get-class (mem (lambda (n) (class-generator)
    ))]
[assume get-var (mem (lambda (c) (/ 1.0 (gamma 1 10)
    )))]
[assume get-std (lambda (c) (sqrt (get-var c)))]
[assume get-mean (mem (lambda (c) (normal 0 (sqrt (*
     10 (get-var c))))))]
[assume u (lambda () (list (get-class 1) (get-class
    2) (get-class 3) (get-class 4) (get-class 5) (
    get-class 6) (get-class 7) (get-class 8) (
    get-class 9) (get-class 10))) ]
[assume K (lambda () (count (unique (u))))]
[observe-csv "dpm-data.csv" (normal (get-mean (
    get-class $1)) (get-std (get-class $1))) $2)]
[predict (u)]
[predict (K)]
```

Figure 3: DP Mixture model expressed in StocPy (left) and Anglican (right)

```
def marsaglia(mean, std):
  x = stocPy.uniform(-1, 1)
  y = stocPy.iuniform(-1, 1)
  s = x*x + y*y
  if s < 1:
    return mean + (std * (x * math.sqrt(-2 * (math.
        log(s) / s))))
  return marsaglia(mean, std)

def marsagliaMean():
  mean = marsaglia(1, math.sqrt(5))
  std = math.sqrt(2)
  stocPy.normal(mean, std, cond=9)
  stocPy.normal(mean, std, cond=8)
  stocPy.observe(mean, name="mean")
```

```
[assume marsaglia-normal (lambda (mu std)
        (begin
          (define x (uniform-continuous -1.0 1.0))
          (define y (uniform-continuous -1.0 1.0))
          (define s (+ (* x x) (* y y)))
          (if (< s 1)
              (+ mu (* std (* x (sqrt (* -2.0 (/ (
                  log s) s))))))
              (marsaglia-normal mu std)))))]
[assume std (sqrt 2)]
[assume mu (marsaglia-normal 1 (sqrt 5))]
[observe (normal mu std) 9]
[observe (normal mu std) 8]
[predict mu]
```

Figure 4: Marsaglia model expressed in StocPy (left) and Anglican (right)

clarity is a subjective perception, but we would argue that the StocPy formulations are friendlier for people without a strong CS or technical background, such as domain experts.